3
Cenário de Referência

Como mencionado no capítulo anterior, o foco desta tese é a caracterização de problemas de desempenho de aplicações construídas sobre tecnologias de *middleware*. No contexto deste trabalho, desempenho é definido em termos de objetivos de nível de serviço (SLO), os quais estão diretamente associados à satisfação do usuário. Um exemplo típico de SLO é um valor limiar para o tempo de resposta de um serviço.

Ainda em relação ao SLO, este trabalho assume que um serviço pode estar em um de dois estados possíveis: *conformidade* ou *violação*. Um serviço é considerado em conformidade com o SLO especificado se o objetivo de desempenho, definido para o mesmo, foi atingido. Caso contrário, o serviço é considerado em violação. Neste trabalho, assume-se que um serviço em violação com os objetivos especificados é um indicativo de baixo desempenho ou degradação de desempenho.

Com o intuito de analisar o comportamento do desempenho de sistemas baseados em *middleware*, um cenário foi fixado para o estudo proposto. Nesse cenário, o SCS (Tecgraf/PUC-Rio, 2010), um sistema de componentes baseados em CORBA, foi adotado como referência para tecnologia de *middleware*. Adicionalmente, uma implementação MapReduce (Dean e Ghemawat, 2004) foi adotada como referência para uma aplicação baseada em *middleware*. Além do cenário de referência, uma arquitetura de gerenciamento, o SMART (Correa e Cerqueira, 2010), foi definida e implementada. Este capítulo introduz o cenário de referência, descrevendo seus componentes principais. Em seguida, discutimos a arquitetura de gerenciamento implementada.

3.1
SCS

Neste trabalho, adotamos como *middleware* de referência um sistema baseado em componentes. Particularmente, o *middleware* escolhido é o SCS (*Software Component System*) (Tecgraf/PUC-Rio, 2010), um sistema de componentes distribuído construído sobre o *middleware* de comunicação CORBA. A compatibilidade do SCS com a versão 2 de CORBA oferece aos componentes

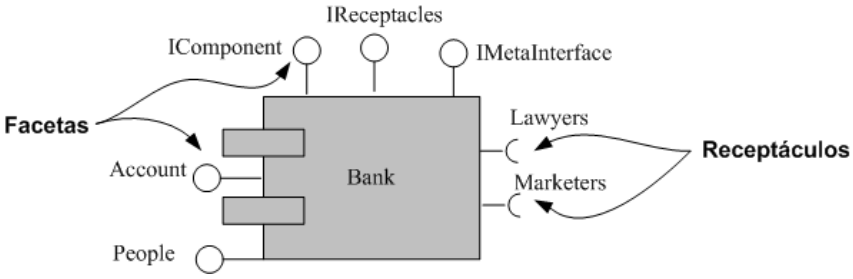
a capacidade de interoperar com outros componentes de terceiros que podem ser desenvolvidos em diferentes linguagens ou plataformas. A infraestrutura de distribuição do SCS é composta por duas partes principais: o modelo de componentes e a infraestrutura de execução.

Como mencionado na Seção 2.2, um modelo de componentes é essencial para um *middleware* baseado em componentes. No SCS, o modelo de componentes define o comportamento comum dos componentes de aplicação e a forma como os mesmos interagem entre si. Nesse último caso, portas servidoras, denominadas *facetas*, e portas clientes, denominadas *receptáculos*, são providas pelo sistema. Além de interação, o modelo de componentes do SCS define também formas de configuração e introspecção. Essas funcionalidades são definidas em três facetas básicas:

- *IComponent*, que permite a identificação, ativação e desativação do componente;
- *IReceptacles*, que gerencia as conexões entre componentes (remotos ou locais); e
- *IMetaInterface*, que provê operações para introspecção.

A Figura 3.1 ilustra um componente SCS. Atualmente, uma implementação do modelo e uma biblioteca de apoio a programação são fornecidas pelo SCS para as linguagens Lua (Ierusalimschy, 2006), Java, C++ e C#. Essas implementações usam, respectivamente, as seguintes implementações CORBA: OiL (Maia, 2009), JacORB, Mico e IIOP.NET.

Figura 3.1: Um componente SCS típico

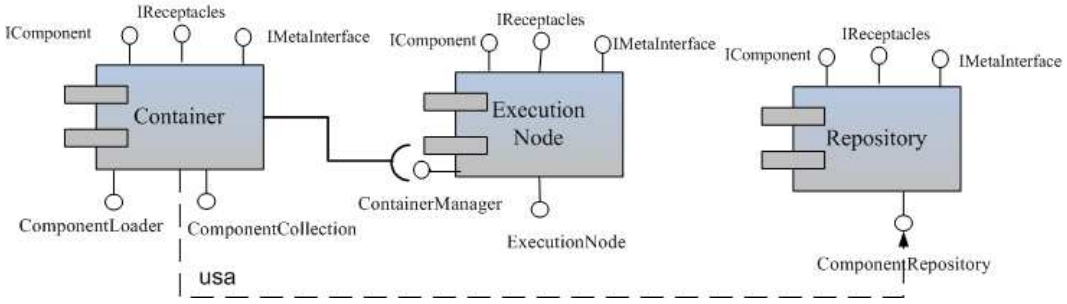


O desenvolvimento de um componente SCS consiste em: *i*) especificar as interfaces em OMG IDL que representarão as facetas e os receptáculos; *ii*) implementar essas interfaces em uma das linguagens suportadas; e *iii*) implementar um código que use o método *newComponent* presente na respectiva biblioteca de apoio. Esse método espera um identificador para o componente e os descritores das facetas e dos receptáculos. Na prática, uma faceta é apenas

um objeto CORBA onde o serviço a ser provido é descrito na forma de interfaces OMG IDL. Por outro lado, um receptáculo é uma estrutura que agrupa um ou mais *proxies* CORBA, os quais referenciam facetas de mesma interface em outros componentes. A associação entre um receptáculo e uma ou mais facetas é representada por uma conexão que é gerenciada pela faceta *IReceptacles*.

O SCS provê também uma infraestrutura de execução para aplicações distribuídas baseadas em componentes (Augusto, 2008). Essa infraestrutura permite a instanciação, configuração, suspensão, interceptação e execução de componentes SCS e é formada pelos seguintes componentes: *Container*, *Execution Node* e *Repository*. Esses componentes são ilustrados na Figura 3.2.

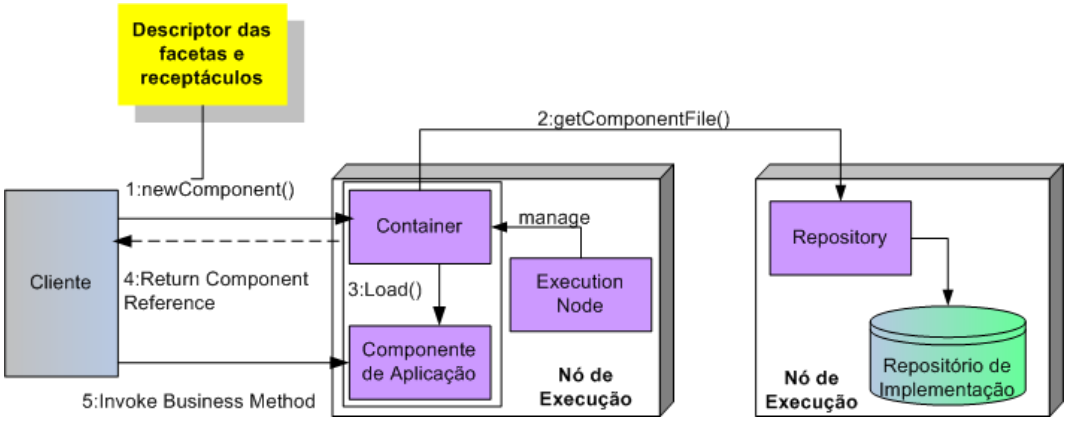
Figura 3.2: Componentes da infraestrutura de execução do SCS



O componente *Container* representa um processo de sistema que disponibiliza um espaço de memória comum a um ou mais componentes SCS. É sua responsabilidade controlar o ciclo de vida dos componentes de aplicação nele hospedados, podendo também monitorá-los, provendo facilidades para interceptação, suspensão e resumo da invocação de métodos remotos. O componente *Execution Node* representa um nó na rede, atuando como porta de entrada na máquina física. Esse componente é responsável por gerenciar a criação dos *Containers* e controlar o acesso desses aos recursos físicos locais. Finalmente, o componente *Repository* armazena implementações de componentes que podem ser publicadas e obtidas remotamente.

A Figura 3.3 ilustra o processo de instanciação de um componente SCS. Nesse processo, o *middleware* provê os serviços de infraestrutura para a execução dos componentes de aplicação, como a configuração, instanciação e carga para uma plataforma específica, além do gerenciamento do ciclo de vida do componente. Os desenvolvedores implementam os componentes de aplicação, fornecendo os descritores das facetas e receptáculos dos componentes.

Figura 3.3: Instanciação de um componente SCS



3.2 O Modelo de Programação MapReduce

MapReduce (Dean e Ghemawat, 2004) é um estilo de programação paralela usado para processar grandes conjuntos de dados em determinados tipos de problemas distribuídos. Nesse modelo de programação, desenvolvedores podem processar conjuntos de dados, escondendo das aplicações detalhes sobre a execução distribuída. Devido a essa característica, o paradigma MapReduce tem ganhado grande popularidade. A principal implementação Java de código aberto, o Hadoop (ApacheSoftwareFoundation, 2007), é usada em grandes corporações, tais como Yahoo! e Facebook, para processar petabytes de dados diariamente (ApacheSoftwareFoundation, 2010).

Essencialmente, o modelo MapReduce permite que usuários escrevam funções *map* e *reduce* em estilo funcional. Uma função *map* recebe um par ⟨chave,valor⟩ como entrada e produz uma lista de pares ⟨chave,valor⟩ como saída. Por outro lado, uma função *reduce* recebe uma chave e uma lista de valores associados como entrada e gera uma lista de novos valores como saída.

$$map :: (key_1, value_1) \rightarrow list(key_2, value_2)$$

$$reduce : (key_2, list(values_2)) \rightarrow list(value_3)$$

Uma aplicação MapReduce é executada de forma paralela através de duas fases. Na primeira fase, todas as operações *map* podem ser executadas independentemente das outras. Na segunda fase, cada operação *reduce* pode depender das saídas geradas pelas operações *map*. Todavia, operações *reduce* podem ser executadas independentemente entre si. A aplicação executa em uma arquitetura mestre-escravo. O fluxo de controle inicia com a aplicação criando várias instâncias do programa em um conjunto de máquinas. Em

seguida, uma das cópias, denominada *Master*, atribui tarefas *map* ou *reduce* para as outras cópias, denominadas *Workers*. A implementação do modelo se encarrega de escalonar essas tarefas de forma que as mesmas executem em recursos distribuídos, controlando problemas como comunicação, paralelismo e tolerância a falhas.

Em resumo, o modelo MapReduce oferece várias vantagens. O programador precisa apenas fornecer as funções *map* e *reduce*, bem como o conjunto de dados de entrada. É responsabilidade da implementação do modelo escalonar as execuções das tarefa de forma transparente e escalável. A seguir, descrevemos brevemente um *framework* MapReduce construído a partir de componentes SCS. Esse *framework* é usado como a aplicação de referência neste trabalho.

3.2.1 Um framework MapReduce usando um Sistema de Componentes

O *framework* MapReduce desenvolvido neste trabalho é similar ao Hadoop em muitos aspectos, especialmente a API de programação. Todavia, o *framework* proposto enfatiza o encapsulamento das funcionalidades providas pelo modelo MapReduce em componentes interoperáveis, os quais podem ser implantados em diferentes plataformas num ambiente distribuído. O *framework* oferece duas implementações do modelo: uma em Java e outra em Lua.

Para definir funções *map* e *reduce*, os usuários implementam as interfaces *Mapper* e *Reducer* respectivamente. A Listagem 3.1 mostra a definição dessas interfaces em Java. Os tipos *Key* e *Value* dos parâmetros das operações *map* e *reduce* representam, respectivamente, a chave e o valor passados para estas funções. Esses tipos são interfaces para as quais a biblioteca de programação oferece algumas implementações, embora o usuário possa também utilizar sua própria implementação. Os parâmetros *collector* e *rpt* representam, respectivamente, o *buffer* de dados para onde o resultado das funções é envidado e o objeto de log usado pela aplicação.

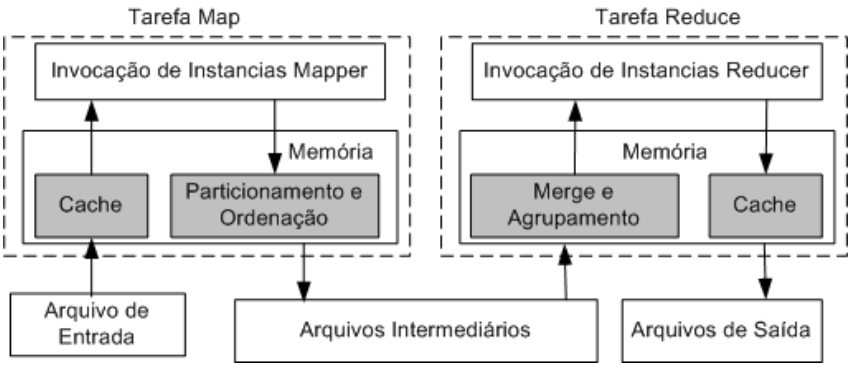
Modelo de Execução

Como ilustrado na Figura 3.4, a execução de uma aplicação MapReduce consiste em dois grandes tipos de tarefas: *Map* e *Reduce*. A execução da aplicação tem início com o particionamento do arquivo de entrada em m fragmentos, os quais serão processados por m tarefas *Map*. Os resultados intermediários gerados pelas tarefas *Map* são particionados em r fragmentos e cada fragmento é processado por uma tarefa *Reduce*.

Listagem 3.1: Interfaces da API de programação

```
1 public interface Mapper {
2   void map(Key key, Value value,
3            OutputCollector collector,
4            Reporter rpt) throws IOMapReduceException;
5 };
6
7 public interface Reducer {
8   void reduce (Key key, Value[] values,
9                OutputCollector collector,
10               Reporter rpt) throws IOMapReduceException;
11 };
```

Figura 3.4: Fluxo de controle de um programa MapReduce



Em cada tarefa *Map*, pares ⟨chave,valor⟩ são extraídos de uma partição do arquivo de entrada e atribuídos a uma função *map* definida pelo usuário. O resultado gerado pela função *map* é coletado para um *buffer* de memória. Esse *buffer* consiste de vários blocos, os quais são distribuídos entre r fragmentos. Uma função *hash* padrão, ou definida pelo usuário, determina o fragmento de destino para um resultado gerado por uma função *map*. Quando uma tarefa *Map* atinge o final da partição do arquivo de entrada que lhe foi atribuída, os dados do *buffer* de memória são ordenados e um arquivo intermediário é gravado para cada um dos r fragmentos. A fase *map* termina quando todas as m tarefas *Map* são executadas.

A fase *reduce* começa combinando os arquivos intermediários. Todos os valores associados com a mesma chave são agrupados e a função *reduce* definida pelo usuário é invocada para executar a operação de redução sobre esses valores. Todos os resultados gerados pela função *reduce* são armazenados em disco.

O modelo de execução do *framework* MapReduce proposto neste traba-

lho adota uma arquitetura mestre-escravo baseada em componentes. O Componente *Master* executa em um contêiner próprio, controlando todo o fluxo de execução da aplicação. Esse componente implementa uma faceta *Master* (Listagem 3.2, linhas 8 a 12), através da qual usuários submetem trabalho. Para tanto, é necessário invocar o método *submitJob*, fornecendo como argumento um arquivo de configuração contendo as seguintes propriedades: localização do arquivo de entrada, os valores de m e r e a identificação dos nós onde os componentes *Master* e *Workers* executam. Componentes *Workers* são instanciados em contêineres individuais em diferentes nós. Esses componentes executam tarefas *Map* e *Reduce*. Um componente *Worker* oferece duas facetas: *Worker* (Listagem 3.2, linhas 14 a 18) e *MapReduceInterface* (Listagem 3.2, linhas 20 a 25). A primeira oferece serviços para o gerenciamento e configuração do componente, enquanto a segunda faceta executa efetivamente uma tarefa. Esta última é representada pela interface *Task* (Listagem 3.2, linhas 2 a 6). A execução da aplicação MapReduce é orquestrada pelo componente *Scheduler*. Esse componente é responsável pela coordenação dos múltiplos recursos disponíveis para realizar a computação da aplicação.

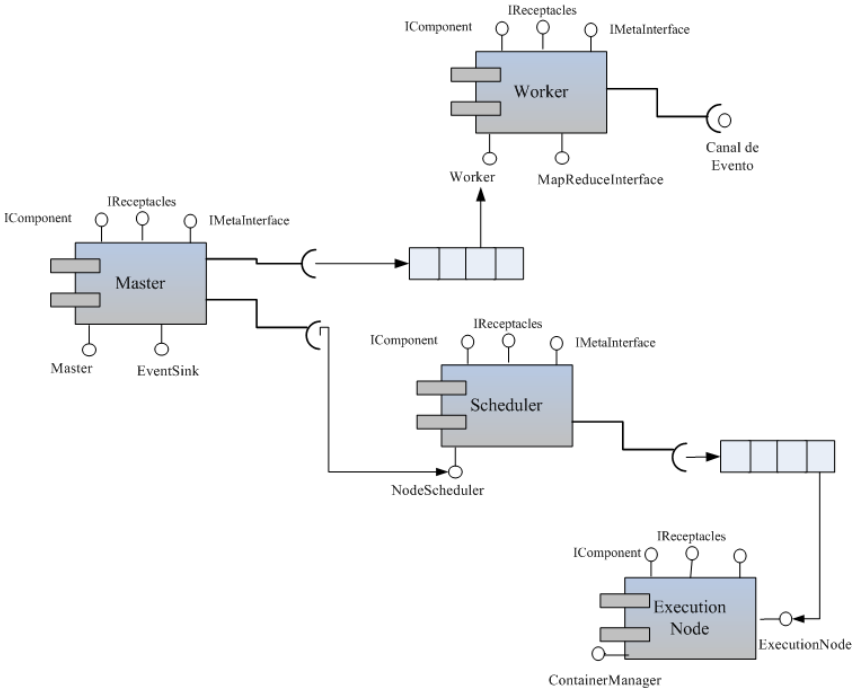
Listagem 3.2: Interfaces do modelo de execução

```
1
2 interface Task {
3   long getId();
4   void setStatus(in TaskStatus status);
5   TaskStatus getStatus();
6 };
7
8 interface Master {
9   void submitJob(in string confFileName)
10    raises (SubmissionException);
11  TaskStatus getMapReducePhase();
12 };
13
14 interface Worker{
15  void execute (in Task t);
16  string getNode();
17  boolean ping();
18 };
19
20 interface MapReduceInterface {
21  void map (in Task t)
```

```
22     raises (IOMapReduceException);
23   void reduce (in Task t)
24     raises (IOMapReduceException);
25 };
```

A Figura 3.5 ilustra o relacionamento entre os componentes do *framework*. Através de um receptáculo, o componente *Master* conecta-se com os componentes *Workers* a fim de lhes atribuir tarefas. Por outro lado, cada componente *Worker* está conectado a um canal de evento, cujo consumidor é o *Master*. Um componente *Worker* usa esse canal para comunicar ao componente *Master* o status final de uma tarefa que lhe foi atribuída. No *Master*, a atribuição de tarefas segue uma política determinada pelo componente *Scheduler*. Este mantém, através de seu receptáculo, uma referência para um conjunto de nós de execução onde *Workers* livres estão disponíveis para novas tarefas.

Figura 3.5: Componentes que formam o framework MapReduce



Como referência a uma aplicação MapReduce, neste trabalho, usamos a aplicação *WordCount*. Essa aplicação conta o número de ocorrências de cada palavra num conjunto de entrada.

Escalonamento

Retomando uma questão essencial no modelo de programação MapReduce, o escalonamento de tarefas é um dos fatores que mais afeta o desempe-

Listagem 3.3: Interface do componente *Scheduler*

```
typedef sequence<execution_node::ExecutionNode>
        ExecutionNodeSeq;

interface NodeScheduler {
  ExecutionNodeSeq sortServers()
    raises (ScheduleException);
};
```

nho das aplicações construídas sobre este modelo. O escalonamento coordena os vários recursos disponíveis para a computação das tarefas *Map* e *Reduce*. Após os usuários submeterem sua aplicação, o escalonador mapeia tarefas *Map* e *Reduce* para diferentes recursos. Ou seja, o desempenho de aplicações MapReduce depende diretamente do algoritmo de escalonamento utilizado.

A listagem 3.3 ilustra a interface *NodeScheduler* oferecida pelo componente *Scheduler*. Essa interface define uma operação *sortServers* que ordena um conjunto de *Workers* livres segundo alguma política de escalonamento. Uma política muito comum em várias implementações é a *alternância circular* ou *Round Robin*. No entanto, algoritmos de escalonamento estático, como o *Round Robin*, não são adequados para ambientes dinâmicos ou de grande escala.

Uma abordagem diferente é ordenar o conjunto de *Workers* pelo *nível de utilização de CPU* nos nós. Neste trabalho, tal política é denominada *Política baseada em CPU*. Apesar de simples e dinâmica, essa política apresenta dois problemas. Primeiro, ela estima o desempenho das aplicações em termos apenas do consumo de CPU. No entanto, aplicações complexas não são limitadas apenas por CPU, mas sim por uma combinação de métricas. Segundo, para funcionar efetivamente, uma política de estimação de desempenho de aplicações deveria levar em consideração não só métricas de sistema, mas também abstrações e métricas da aplicação (Cohen et al., 2004).

A abordagem de escalonamento adotada para o *framework* MapReduce proposto neste trabalho é ordenar o conjunto de *Workers* pela probabilidade de um nó violar um objetivo especificado ou, mais precisamente, um SLO. Tal política será denominada de *Política baseada na Probabilidade de Violação*. Essa probabilidade, por sua vez, é estimada em função de diversas métricas de sistema e também de aplicação. Uma arquitetura de gerenciamento, a qual é descrita a seguir, cria e mantém o modelo que mapeia um conjunto de métricas observadas em um estado de um serviço.

3.3 SMART

Tendo definido o cenário de referência, a investigação, a fim de responder as questões $\mathbf{Q_1}$, $\mathbf{Q_2}$, $\mathbf{Q_3}$ e $\mathbf{Q_4}$ colocadas no Capítulo 1 deste trabalho, prosseguiu com a definição e implementação de uma arquitetura de gerenciamento de sistemas. Essa arquitetura foi denominada SMART (*Self-MAnaged Resource uTilization*).

A Figura 3.6(a) mostra uma visão geral da arquitetura de gerenciamento proposta neste trabalho. No SMART, uma máquina de inferência executa em cada nó da rede onde um componente de aplicação é instanciado. A máquina de inferência analisa uma coleção de dados coletados pela infraestrutura de monitoramento do SCS (Andrea, 2009) e periodicamente: *i*) constrói um modelo de comportamento do desempenho da aplicação; *ii*) estima, com certa robustez, o estado de um serviço; e *iii*) se necessário, faz o diagnóstico da causa de um problema de desempenho. Cada nó é analisado separadamente dos outros nós.

A Figura 3.6(b) ilustra essa arquitetura em termos de suas camadas: a camada de aplicação, a infraestrutura de monitoramento e a máquina de inferência. A camada de aplicação compreende o ambiente de execução no qual as aplicações executam. O sistema de componentes, através dos serviços representados pelo nó de execução e os contêineres, gerencia essa camada, controlando as condições nas quais os componentes são implantados e executados. A seguir, a infraestrutura de monitoramento e a máquina de inferência são descritas.

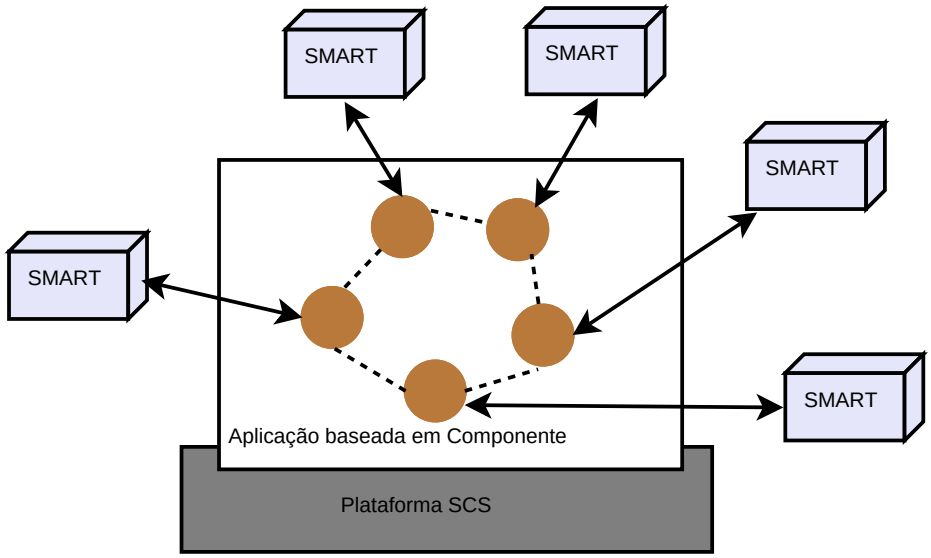
3.3.1 Infraestrutura de Monitoramento

O SCS provê uma infraestrutura de coleta e publicação de dados (ou métricas) sobre os componentes de aplicação e o ambiente em que os mesmos executam. De maneira geral, esses dados são coletados através de quatro formas distintas:

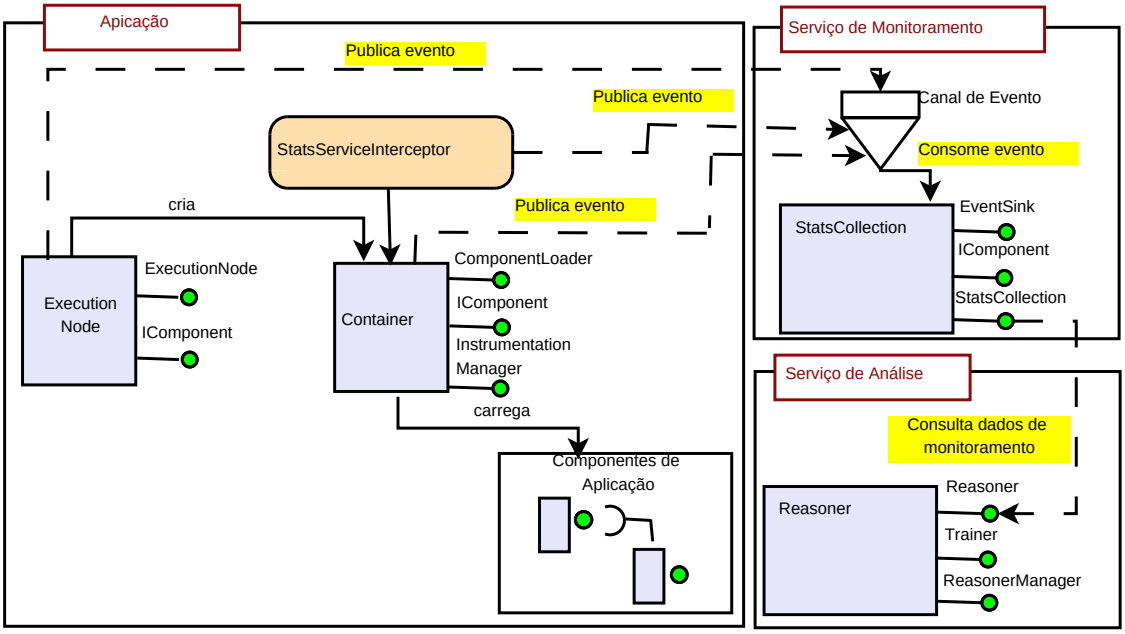
- *Daemon* no componente *Execution Node*: o componente *Execution Node* executa uma *thread* que periodicamente coleta dados sobre a carga aplicada sobre os recursos físicos da máquina. Isso inclui diversos dados sobre utilização de recursos referentes a CPU, memória, disco e rede.
- *Daemon* no componente *Container*: similarmente, cada componente *Container* inicializa um *daemon* que é responsável por coletar periodicamente dados sobre a utilização de CPU daquele processo.

Figura 3.6: A arquitetura do SMART.

(a) Visão geral do SMART



(b) Camadas do SMART



– Interceptação no componente *Container*: dados referentes aos componentes de aplicação são coletados através de interceptadores (componente *StatsServiceInterceptor* da Figura 3.6(b)) instalados nos componentes *Container*. Esses interceptadores permitem a inserção de código em diferentes pontos da cadeia de invocação de um método remoto e a computação de métricas como o tempo de resposta e a vazão dos serviços oferecidos por um componente.

– Metadados providos pela estrutura de execução do SCS, os quais fornecem informação sobre a distribuição dos componentes nos vários nós da rede.

Em resumo, a infraestrutura de monitoramento do SCS permite coletar dados de sistema (como dados de utilização de CPU, memória, disco e rede), do *middleware* (como utilização de CPU nos contêineres) e da aplicação (como o tempo médio de resposta de um método definido em um componente de aplicação). Além desses dados, os desenvolvedores podem também definir suas próprias métricas a fim de atender interesses de aplicações específicas.

Uma vez coletados, os dados são publicados imediatamente em canais de eventos que têm como consumidor um ou mais componentes mediadores (componente *StatsCollection* da Figura 3.6(b)). Esses componentes recebem os dados ou métricas coletadas nos diferentes elementos da arquitetura e os agrega por elemento. Dessa forma, para cada nó ou máquina da rede, é possível obter métricas de desempenho referentes aos recursos físicos do próprio nó, dos contêineres instanciados nessa máquina e dos componentes e métodos carregados nos contêineres. Os componentes mediadores oferecem métodos para consultar tais dados.

Retomando o cenário de referência considerado neste trabalho, a Tabela 3.1 lista os dados coletados pela infraestrutura de monitoramento quando a aplicação *WordCount* é executada. Os dados da tabela são coletados a cada 15 segundos. As quatro primeiras métricas estão relacionadas ao nó de execução e representam, respectivamente, o número de bytes recebidos da rede, o número de bytes transmitidos pela rede, o número de bytes lidos do disco e o número de bytes escritos no disco, numa janela de 15 segundos. Essas métricas são monitoradas com o intuito de estimar a carga de rede e disco a qual o nó está submetido. *nfs_bytes_read* e *nfs_bytes_write* são também métricas relacionadas ao nó de execução, monitoradas a fim de aferir a carga de operações de entrada e saída provenientes do sistema de arquivo em rede. Duas outras métricas completam a instrumentação no nó de execução: *cpu_usage* e *memory_usage*. Essas métricas guardam, respectivamente, o uso de CPU e de memória na máquina, durante uma janela de monitoramento.

Além das métricas do nó de execução, métricas relacionadas ao contêiner também são coletadas. *containers_avg_cpu_usage* é coletada com o intuito de estimar a utilização média de CPU pelo processo representado pelo contêiner. Por outro lado, a métrica de aplicação *map_reduce_phase* é coletada a fim de determinar a fase, Map ou Reduce, em que se encontra a aplicação em um dado momento. A figura 3.7, ilustra a distribuição dos tempos de resposta das operações *map* e *reduce*, quando a aplicação *WordCount* processa um arquivo

de 1 GB. É possível notar que, na grande maioria das vezes, o tempo de resposta da operação *reduce* é significativamente menor que o tempo de resposta da operação *map*. Dessa forma, torna-se essencial identificar a fase na qual se encontra a aplicação quando a métrica *avg_response_time*, que guarda o valor médio do tempo de resposta de uma operação numa janela de monitoramento, é coletada. A métrica *map_reduce_phase* é responsável por tal identificação.

Por fim, é importante ressaltar que a escolha do valor de 15 segundos para a janela de monitoramento se deve ao fato desse intervalo ter se mostrado suficiente para observar tanto eventos de conformidade como de violação de objetivos de desempenho. Adicionalmente, como demonstrado pelos autores em (Fonseca et al., 2008), essa frequência de medição, ou coleta de dados, implica em uma sobrecarga ao sistema de menos de 4%.

Tabela 3.1: Métricas coletadas e usadas no processo de análise

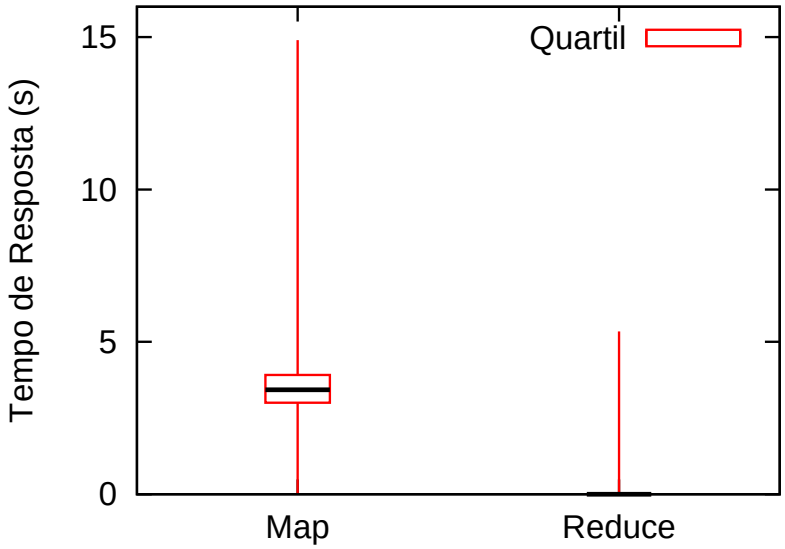
Métricas do nó de execução	
net_bytes_in	Número de bytes recebidos da rede
net_bytes_out	Número de bytes transmitidos pela rede
disk_bytes_read	Número de bytes lidos do disco
disk_bytes_write	Número de bytes escritos para o disco
nfs_bytes_read	Numero de bytes lidos de nfs
nfs_bytes_write	Número de bytes escritos para nfs
cpu_usage	Tempo de CPU (system+user)
memory_usage	Quantidade de memória utilizada
Métricas do contêiner	
containers_avg_cpu_usage	% média de CPU
map_reduce_phase	Fase em que se encontra a aplicação (map ou reduce)
Métricas do método	
avg_response_time	Tempo médio de resposta de uma operação (map ou reduce)

3.3.2 Máquina de Inferência

A máquina de inferência do SMART é responsável pela análise dos dados coletados pela infraestrutura de monitoramento. Basicamente, a análise consiste em encontrar uma correlação entre valores de métricas monitoradas e estados dos serviços, ou seja, conformidade ou violação de um SLO. No SMART, essa correlação é tratada como um problema de classificação.

Um classificador é uma função que atribui um rótulo ou classe a um objeto descrito por uma ou mais propriedades. Tipicamente, um problema

Figura 3.7: Distribuição do tempo de resposta para as operações map e reduce



de classificação envolve duas fases. Na fase de treinamento ou aprendizado, um classificador é construído a partir de um conjunto de dados ou exemplos rotulados. Na fase de classificação, o classificador construído é usado para estimar o rótulo ou classe de um novo objeto.

No SMART, a máquina de inferência é implementada por um componente SCS denominado *Reasoner* (Figura 3.6(b)). Esse componente implementa duas facetas principais, como mostrado na Listagem 3.4. A faceta *Trainer* (Listagem 3.4, linhas 44 a 47) implementa o comportamento do componente na fase de treinamento, criando o arquivo de treinamento que será usado na fase de aprendizado do classificador. Essencialmente esse comportamento consiste em:

1. interagir com o componente mediador em intervalos regulares de tempo;
2. obter as métricas coletadas referentes ao nó de execução, os contêineres nele criados e os componentes instanciados (Tabela 3.1) e
3. gravar as métricas obtidas em um arquivo de *log*.

Por outro lado, a faceta *Reasoner* (Listagem 3.4, linhas 38 a 42) implementa o comportamento do componente na fase de classificação. Esse comportamento consiste em:

1. construir o classificador a partir do arquivo de treinamento;
2. em intervalos regulares de tempo, obter os valores correntes para as métricas de desempenho monitoradas e

3. classificar o estado de um serviço a partir da amostra obtida.

Na faceta *Reasoner*, o método *classifySystemState* pode ser invocado para obter os dados resultantes da análise do comportamento da aplicação. Esses dados são retornados na forma da estrutura *ClassificationResult* (Listagem 3.4, linhas 26 a 30) e consistem no estado do serviço (conformidade ou violação), na probabilidade do serviço violar um SLO e, em caso de violação, nas métricas monitoradas, ordenadas de forma ascendente pelo grau de influência sobre o problema observado.

No SMART, um classificador é definido através do descritor *Classifier-Description* (Listagem 3.4, linhas 3 a 7). Este descritor define o classificador em termos do nome da classe que implementa o algoritmo de classificação, as opções de configuração que são usadas pelo algoritmo e o nome da classe que implementa o algoritmo de *ranking*. Este último é responsável por classificar as métricas monitoradas pelo grau de influência que as mesmas exercem em um problema de desempenho. Analogamente, uma métrica também é definida através de um descritor. A estrutura *Scheme_str* (Listagem 3.4, linhas 9 a 16) é usada com esse propósito. A definição de uma métrica contém os metadados necessários para a máquina de inferência interpretar e processar a métrica em questão. Para processar uma métrica, o SMART necessita dos seguintes metadados: o nome da métrica, o intervalo de discretização aplicável ao domínio da variável (métrica), a indicação que determina se a métrica é a variável alvo da classificação, a indicação que determina se a métrica é usada para especificar o objetivo de desempenho (SLO) e a indicação que determina se a métrica deve ser considerada no processamento do algoritmo de *ranking*. Quando uma métrica é instanciada, a estrutura *Instance_str* (Listagem 3.4, linhas 18 a 21) associa o valor coletado pelo serviço de monitoramento à definição da métrica. O esquema contendo a definição de todas as métricas usadas pela máquina de inferência deve ser fornecido pelo usuário. Para isso, o usuário deve implementar a interface *Filter* (Listagem 3.4, linhas 32 a 36) a qual define dois métodos. O primeiro método, *filterMetrics*, é usado para filtrar os dados coletados pelo serviço de monitoramento do SCS e instanciar as métricas analisadas pela máquina de inferência. O segundo método, *getMetricDescriptions*, retorna a definição de todas as métricas usadas pela máquina de inferência.

Listagem 3.4: Interfaces *Trainer* e *Reasoner* do componente *Reasoner*

```
1  enum SystemStatus{DEGRADATION,COMPLIANCE};
2
3  struct ClassifierDescription {
4      string name;
```

```
5       sequence<string> options;
6       string ranker;
7   };
8
9   struct Scheme_str {
10      string metricName;
11      long numberOfBins;
12      boolean classVariable;
13      boolean sloBased;
14      boolean rankable;
15      long metricIndex;
16  };
17
18  struct Instance_str {
19    long metricIndex;
20    double metricValue;
21  };
22
23  typedef sequence<Scheme_str> SchemeSeq;
24  typedef sequence<Instance_str>InstanceSeq;
25
26  struct ClassificationResult {
27    SystemStatus status;
28    double degradationProbability;
29    SchemeSeq culprit;
30  };
31
32  interface Filter {
33    InstanceSeq filterMetrics (
34                    in scs::instrumentation::MachineStats stats);
35    SchemeSeq getMetricDescriptions();
36  };
37
38  interface Reasoner{
39    void startReasoning() raises (ReasoningFailure);
40    void stopReasoning();
41    ClassificationResult classifySystemState();
42  };
43
```

```
44  interface Trainer{
45    void startTraining() raises (TrainingFailure);
46    void stopTraining();
47  };
```

Além das facetas descritas acima, o componente *Reasoner* possui também uma faceta de gerenciamento, através da qual configura-se a máquina de inferência. É possível configurar o modo de operação do componente (treinamento ou análise), a periodicidade em que ele interage com o componente mediador para coleta de dados, o nome do arquivo de treinamento (a ser gerado pela faceta *Trainer* ou lido pela faceta *Reasoner*), o valor limiar do SLO e o algoritmo que será usado na construção do classificador e do algoritmo de *ranking*. A métrica a partir da qual será definido o SLO e o seu limiar são usados por um módulo supervisor para rotular os dados do arquivo de treinamento. Esse módulo é implementado por uma classe Java responsável por todo o pré-processamento dos dados. A metodologia usada na rotulação dos dados e os algoritmos de classificação usados neste trabalho são descritos no próximo capítulo.

3.4 Considerações Finais

Neste capítulo, foi descrito o cenário de referência usado no estudo proposto. Foi descrito também o SMART, a arquitetura de gerenciamento de sistemas implementada com o intuito de validar algumas técnicas de aprendizado estatístico para caracterização de problemas de desempenho. Vale ressaltar algumas limitações do cenário estabelecido como referência para este trabalho. Com relação às aplicações que usam o modelo de programação MapReduce, pode-se afirmar que esse tipo de aplicação apresenta um comportamento bastante homogêneo em termos de mistura de transações. Basicamente, apenas dois tipos de operações são executados (*map* e *reduce*) e ambos os tipos ocorrem de forma excludente na aplicação, ou seja, operações do tipo *reduce* nunca são executadas concorrentemente com operações do tipo *map*.

Quanto ao SMART, vale destacar que a máquina de inferência correlaciona apenas dados locais, não tratando ou analisando situações que possam tirar proveito de correlações entre eventos distribuídos. O SMART, ou melhor, a instanciação do SMART usada neste trabalho, também só trata uma aplicação ou várias aplicações compostas por apenas um tipo de processo. Como consequência, a arquitetura é capaz de gerenciar apenas processos que tenham o mesmo perfil de uso de recursos. Apesar dessas limitações, no próximo capítulo, o cenário de referência e a arquitetura de gerenciamento são usados para ava-

liar a aplicabilidade das principais famílias de classificadores estatísticos para caracterizar um problema de desempenho.